Introduction to C++

Georgia Koutsandria

Internet of Things A.Y. 19-20

Prof. Chiara Petrioli
Dept. of Computer Science
Sapienza University of Rome

How to compile a C++ program

- **Windows**: Install an Integrated Development Interface (IDE).
  - Dev-C++ http://www.bloodshed.net/dev/index.html

- **Mac:** Install Xcode with the gcc/clang compilers.

```
g++ -std=c++11 example.cpp -o example_program OR
clang++ -std=c++11 -stdlib=libc++ example.cpp -o example_program
```

- **Linux:** Compile your code directly from the terminal using the following commnad `g++ -std=c++0x example.cpp -o example_program`

Associative Containers in the C++ standard library (Recap)

Associative Containers

- Elements are stored and retrieved by a key.
- Two primary associative container types: `map` and `set`.
- The C++ library provides eight associative containers.

Associative Container Types

Container Type	
`map`	Holds key-value pairs
`set`	The key is the value
`multimap`	A key can appear multiple times
`multiset`	A key can appear multiple times
`unordered_map (c++11)`	Organized by a hash function
`unordered_set (c++11)`	Organized by a hash function
`unordered_multimap(c++11)`	Hashed map; keys can appear multiple times
`unordered_multiset(c++11)`	Hashed set; keys can appear multiple times

The map associative container

- A collection of (key, value) pairs; often referred to as an associative array.
- Values are found by a key rather than by their position (as in arrays).
- E.g.: Mapping names to phone numbers; Each pair contains a person's name as a key and a phone number as its value.

```
#include <map>
```

```
map<key, value> name;
```

The set associative container

- It store unique elements following a specific order.
- The value of an element is its *key;* it must be unique.
- The value of the elements cannot be modified once in the container.
- The value of the elements can be either inserted or removed from the container.

```
#include <set>
```

```
set<key> name;
```

Range-based Loop

Range-based loop

- A more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container (array, vector, map, set, etc.).
- For observing elements in a container. i.e., read-only:
  1. If the objects are cheap to copy (capture by value)

```
for (auto elem : container_name)
```

  2. Capture by const reference

```
for (const auto& elem : container_name)
```

    - When modifying the elements in the container:
      - Capture by non-const reference

```
for (auto& elem : container_name)
```

Loop through Map

```
map<string,int>::iterator it;

for(it=myMap.begin();it!=myMap.end();it++)
{
        cout << it->first << ": "
             << it->second
             << endl;
}
```

```
for(auto const& x : myMap)
{
        cout << x.first << ": "
             << x.second
             << endl;
}
```

`auto` : Tells the compiler to deduce the type of a declared variable from its initialization expression.

Loop through Set

```
set<string,int>::iterator it;

for(it=mySet.begin();it!=mySet.end();it++)
        cout << *it << endl;
```

```
for(auto elem : mySet)
        cout << elem << " , ";
```

`auto` : Tells the compiler to deduce the type of a declared variable from its initialization expression.

Exercise 1

Write a program that initializes a set which contains 5 integers and prints the contents of the set container. Use two different ways to loop through the set: 1) Using an iterator; 2) Using type inference (auto).

Exercise 1: Solution

```cpp
#include <iostream>
#include <set>
using namespace std;
int main(){
    int mynumbers[5] = {23, 10, 45, 5, 3};
    set<int> myset (mynumbers, mynumbers + 5);
    cout << "myset contains: ";
    for(set<int>::iterator iter = myset.begin() ;iter != myset.end();iter++)
       cout << " " << *iter;
    cout << endl;
    cout << "myset contains: ";
    for(auto elem : myset)
       cout << " " << elem;
    cout << endl;
    return 0;
}
```

Functions (Cont.)

Passing arguments to a function

- Arguments can be passed *by value*; Only copies of the variables values at that moment are passed to thefunction; Modifications on the values of the variables have not effect on the values of the variables outside the function.

- Arguments can be passed *by reference*; The variable itself is passed to the function; Any modifications on the local variables within the function are reflected in the variables passed as arguments in the call

Passing arguments to a function (Cont.)

- Passing arguments *by const reference*. **Why**?
- Passing *by value* requires that all arguments are copied into the function parameters.->time consuming when handling large structs, classes, etc.
  - *Solution*: arguments are passed *by reference.*
  - *Problem:* Undesirable when we want read-only arguments.
    - *(More appropriate) solution:* pass *by const reference*
      - *Minimum performance penalty (not copying arguments)*
      - *Function cannot change the value of the arguments.*

Passing arguments by const reference: An (wrong)example

```
//passing parameters by const reference
#include <iostream>
using namespace std;
void foo(const int &a){
      a = 2;          !
}
```

Compiler will complain!

A const reference cannot have its value changed!

Passing arguments to a function

Q: Can we pass an entire array as an argument to a function?

Passing arguments to a function

Q: Can we pass an entire array as an argument to a function?

A: Not directly but «indirectly!»

Passing arguments to a function

- While an entire array cannot be passed as an argument to a function, pointers to an array can.
- There are different ways to do so:
  1. Formal parameter as a pointer:

```
void function_name(type *param){}
```

  2. Formal parameter as a sized array:

```
void function_name(type param[n]){}
```

  3. Formal parameter as an unsized array:

```
void function_name(type param[]){}
```

Passing arguments to a function
An example (Case 1)

```
#include <iostream>
using namespace std;
double calcAverage(int *arr, int size);
int main(){
        int numbers[5] = {2, 4, 6, 8};
        double avg;
        // int * p = numbers;
        avg = calcAverage(numbers, 4);
        cout << "Average is: " << avg << endl;
        return 0;
}
                double calcAverage(int *arr, int size){
                        int sum = 0;
                        double k;
                       for (int n=0;n<size;n++)
                              sum +=arr[n];
                       k = double(sum)/size;
                        return k;

                }
```

Passing arguments to a function
An example (Case 2)

```
#include <iostream>
using namespace std;
double calcAverage(int arr[], int size);
int main(){
        int numbers[5] = {2, 4, 6, 8};
        double avg;
        avg = calcAverage(numbers, 4);
        cout << "Average is: " << avg << endl;
        return 0;
}
                double calcAverage(int arr[], int size){
                        int sum = 0;
                        double k;
                       for (int n=0;n<size;n++)
                              sum +=arr[n];
                       k = double(sum)/size;
                        return k;

                }
```

Passing arguments to a function
An example (Case 3)

```
#include <iostream>
using namespace std;
double calcAverage(int arr[5], int size);
int main(){
        int numbers[5] = {2, 4, 6, 8};
        double avg;
        avg = calcAverage(numbers, 4);
        cout << "Average is: " << avg << endl;
        return 0;
}
                double calcAverage(int arr[5], int size){
                        int sum = 0;
                        double k;
                       for (int n=0;n<size;n++)
                              sum +=arr[n];
                       k = double(sum)/size;
                        return k;

                }
```

Data Structures

Data Structures

- A group of data elements of different kinds grouped together under a single name.
- Data elements (*members)* can be of different types and lengths.

```
struct type_name{
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
    .
    .
    .
}object_names;
```

Defining data structures

- Keyword "struct" is used to create the structure.
- `type_name:` The name of the structure type.
- `member_name:` The name of the data member.
- `object_names:` A set of valid identifiers for objects that have the type of this structure.

```
struct type_name{
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
    .
    .
    .
}object_names;
```

Defining data structures: An example

It declares a structure type, called product

```
struct product{
    int weight;
    double price;
};
```

Two members, each of a different type

```
product apple;
product melon,orange;
```

Three objects of structure type are declared.

Defining data structures: An example (Alternative option)

structure type name

```
struct product{
    int weight;
    double price;
};

product apple;
product melon,orange;
```

Objects of type product

structure type name

```
struct product{
    int weight;
    double price;
} apple, melon, orange;
```

Name objects can be used to directly declare objects of the structure type.

Accessing the members

- Once a member is declared, it can be accessed directly.
- Syntax: Insert a dot (.) between the object name and the member name.
- E.g.: Each of the objects has a data type corresponds to the member it refers to.
  - apple.weight
  - apple.price
  - melon.weight
  - melon.price
  - orange.weight
  - orange.price

**.weight* are of type int

**.price* are of type double

Initializing structure members

- Structure members can be initialized using curly braces, i.e., {}.

```
#include <iostream>
using namespace std;
struct point{
    int x, y;
};
int main (){
      point p1 = {0,1};
      cout << "Printing x coordinate of p1: " << p1.x << "\n";
      cout << "Printing y coordinate of p1: " << p1.y << "\n";
      return 0;

}
```

Initializing structure members

- Structure members can be initialized using curly braces, i.e., {}, or with declaration.

```
#include <iostream>
using namespace std;
struct point{
    int x = 0;
    int y = 1;
};
int main (){
      point p1;
      cout << "Printing x coordinate of p1: " << p1.x << "\n";
      cout << "Printing y coordinate of p1: " << p1.y << "\n";
      return 0;

}
```

Array of structures

- We can create an array of structures. Each array will have the same structure members.

```
#include <iostream>
using namespace std;
struct student{
    int studentId;
    string firstName, lastName;
};
int main (){
      student stud[2];
      for(int i=0;i<2;i++){
            cout << "Enter the id of the student:";
            cin >> stud[i].studentId;
            cout << "Enter the first name of the student :";
            cin >> stud[i].firstName;
            cout << "Enter the last name of the student :" << endl;
            cin >> stud[i]. lastName;}
      return 0;
}
```

Data structures and functions

- Structure elements can be passed to a function as normal agruments.
    1. by value
        - The values of the elements are passed to the function.
        - The entire structure can be passed to a function.
    2. by reference
        - The address of the structure element is passed to the function.
- Structure elements can be returned from a function as normal arguments.

Data structures and functions

```
struct product{
    int weight;
    double price;
} apple;
```

Individual elements are passed in a function

```
void func1(apple.weight, apple.price){}
```

Data structures and functions

- The entire structure can be passed to a function by value.
- Any changes to the contents of the structure inside the function, do not affect the structure itself.

```
struct product{
    int weight;
    double price;
} apple;
```

Entire structue is passed to a function

```
void func1(product fruit){}
```

Data structures and functions: An example

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
struct movies_t{
  int year;
  string title;
}mine, yours;

void printmovie (movies_t movie){
      cout << movie.title;
      cout << " (" << movie.year << ")"
           << endl;
}
```

```
int main(){
  string mystr;
  mine.title = "Goodbye Bafana";
  mine.year = 2007;
  cout << "Enter a title: ";
  getline(cin, yours.title);
  cout << "Enter year: ";
  getline(cin, mystr);
  stringstream(mystr) >> yours.year;
  cout << "My favorite movie is: ";
  printmovie(mine);
  cout << "Your favorite movie is: ";
  printmovie(yours);

  return 0;
}
```

Pointers to Structures

Pointers to Structures

- A structure can be pointed to by its own type of pointers.

```
struct movies_t{
  int year;
  string title;
};

movies_t amovie;
movies_t  * pmovie;
pmovie = &amovie;
```

An object of structure type movies_t

A pointer that points to objects of structure type movies_t

The value of the pointer pmovie is assigned the address of object amovie.

Pointers to Structures

- The arrow operator (->) is a dereference operator that is used exclusiveley with pointers to objects that have members; It allows access to the member of an object directly from its address.

Expression	What is evaluated	Equivalent
`a.b`	Member b of object a	
`a–>b`	Member b of object pointed to by a	`(*a).b`
`*a.b`	Value pointed to by member b of object a	`*(a.b)`

Pointers to Structures: An example

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
struct movies_t{
  int year;
  string title;
};
int main(){
  string mystr;
  movies_t amovie;
  movies_t  * pmovie;
  pmovie = &amovie;
  cout << "Enter a title: ";
  getline(cin, pmovie->title);
  cout << "Enter year: ";
  getline(cin, mystr);
  stringstream(mystr) >> pmovie->year;
  cout << "You have entered: " << pmovie->title;
  cout << " (" << pmovie->year << ")" << endl;
  return 0;
}
```

Pointers to Structures: An (other) example

```
#include <iostream>
using namespace std;
struct movies_t{
  int year;
  string title;
}mine;
void printmovie (movies_t *movie){
       cout << movie->title;
       cout << " (" << movie->year << ")" << endl;
}
int main(){
  mine.title = "Goodbye Bafana";
  mine.year = 2007;
  cout << "My favorite movie is: ";
  printmovie(&mine);
  return 0;
}
```

Nesting Structures

Nesting Structures

- Structures can be nested in such a way that an element of a structure is itself another structure.

```
struct movies_t{
  int year;
  string title;
};

struct friends_t{
  int year;
  string name;
  string email;
  movies_t favorite_movie;

}gina, gabriele;

friends_t * pfriends = &gina;
```

```
gina.name
gabriele.favorite_movie.title
gina.favorite_movie.year
pfriends->favorite_movie.year
```

Exercise 1

Write a program that implements a structure **array** to construct a database for the products of a supermarket. Your program should take as input the name and the price of 5 products (from the keyboard/user) and it should display them on the screen in a table manner.

- product: a data structure.
- pr : an array structure/object of size 5.
- name: member to store the name of the product.
- price: member to store the price of the product.

```
Enter the name of product 1: Milk
Enter the price of product 1: 0.9

Enter the name of product 2:Shampoo
Enter the price of product 2:5.23

Enter the name of product 3:Water
Enter the price of product 3:0.5

Enter the name of product 4:Bread
Enter the price of product 4:1.99

Enter the name of product 5:Sugar
Enter the price of product 5:2.34

Product Name            Price (Euro)
Milk                    0.9
Shampoo                 5.23
Water                   0.5
Bread                   1.99
Sugar                   2.34
```

Exercise 1-Solution

```
#include <iostream>
using namespace std;
struct product{
    char name[20];
    float price;
} pr[5];

int main(){
   for(int i=0;i<5;i++){
       cout << "Enter the name of product " << i+1 << ":";
       cin >> pr[i].name;
       cout << "Enter the price of product " << i+1 << ":";
       cin >> pr[i].price;
       cout << endl;
    }
    cout << "Product Name" << "\t \t" << "Price (Euro)" << endl;
    for(int i=0;i<5;i++)
       cout << pr[i].name << "\t \t \t" << pr[i].price << endl;

    return 0;
}
```

Classes

Classes

- `class_name:` A valid identifier for the class.
- `object_names:` An optional list of names for objects;
  An object is an instantiation of a class.
- `members:` Contained in the body of the declaration; can be data or function declarations.
- `access_specifiers:` Modify the access rights for the members of the class (optional).

```
class class_name{
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  . . .
}object_names;
```

Access specifier

- **Private**: Accessible only from within other members of the same class (default).
- **Protected**: Accessible from other members of the same class and also from members of their derived classes.
- **Public**: Accessible from anywherhe where the object is visible.

Classes: An example

Class declaration

Name of class

```
class Rectangle{
    int width, height;
  public:
    void set_values(int,int);
    int area(void);
}rect;
```

Class contains four members

Two data members of type int; private access

Two member functions; public access. Only the declaration is included.

An object, i.e., a variable, of the class

Class vs. Object name

- Rectangle: The class name
- rect: An object of type Rectangle
- Analogy:

```
int a;
```

The type name (the class)

The variable name (the object)

```
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(void);
}rect;
```

Accessing public members of a class

- Public objects can be accessed as if they were normal functions or variables.

- Use of dot (.) between object name and member name.

- E.g.:
```
rect.set_values(3,4);
myarea = rect.area();
```

```
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(void);
}rect;
```

Accessing members of a class

```
class Rectangle{
    int width, height;
  public:
    void set_values(int,int);
    int area(void);
}rect;
```

!

Members with private access cannot be accessed from outside of the class. They can only be referred to from within other members of the same class.

Q: What would happen if your program tries to access a private data member from outside of a class?

An example

```
#include <iostream>
using namespace std;

class MyClass{
      int var1, var2;
};

int main()
{
  MyClass mc;
  mc.var1 = 10;
  cout << "var1: " << mc.var1 << endl;
  return 0;
}
```

Q: What would happen if your program tries to access a private data member from outside of a class?

A: Compilation will fail! You will get the following error:

```
Georgias-MacBook-Pro:C++ examples gina$ g++ -std=c++11 classes.cpp -o classes
                    error:
  mc.var1 = 5;
      ^
                  note: implicitly declared private here
int var1, var2;
    ^
                    error:
  cout << "var1: " << mc.var1 << endl;
                             ^
                  note: implicitly declared private here
int var1, var2;
    ^
2 errors generated.
```

Defining a member function

1. Within the class definition: Function is automatically considered an inline member function by the compiler.

2. Include declaration and define it later outside the class: A normal (not-inline) class member function.

An example

The scope operator (::) is used in the definition of a class member to define a member of class outside the class itself.

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(){return width*height;}
};
void Rectangle::set_values(int x, int y){
      width = x;
      height = y;
}
int main(){
  Rectangle rect;
  rect.set_values(3,4);
  cout << "area: " << rect.area() << endl;
  return 0;
}
```

The scope operator(::)

- It specifies the class to which the member being defined belongs.
- It grands exactly the same scope properties as if this function definition was directly included within the class definition.

Multiple object declaration

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(){return width*height;}
};
void Rectangle::set_values(int x, int y){
       width = x;
       height = y;}
int main(){
  Rectangle rect, rectb;
  rect.set_values(3,4);
  rectb.set_values(5,6);
  cout << "area: " << rect.area() << endl;
  cout << "areab: " << rectb.area() << endl;
  return 0;
}
```

Two instances (objects)

Q: What would happen in the previous example if we called the member function `area` before having called `set_values`?

Q: What would happen in the previous example if we called the member function `area` before having called `set_values`?

A: An undetermined result, since the members `width` and `height` had never been assigned a value.

Q: What would happen in the previous example if we called the member function `area` before having called `set_values`?

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(){return width*height;}
};
void Rectangle::set_values(int x, int y){
       width = x;
       height = y;}
int main(){
  Rectangle rect, rectb;
  cout << "area: " << rect.area() << endl;
  rect.set_values(3,4);
  rectb.set_values(5,6);
  cout << "areab: " << rectb.area() << endl;
  return 0;
}
```

Q: What would happen in the previous example if we called the member function `area` before having called `set_values`?

A: An undetermined result, since the members `width` and `height` had never been assigned a value.

```
Georgias-MacBook-Pro:C++ examples gina$ g++ -std=c++11  rectangleError.cpp -o rectangleError
Georgias-MacBook-Pro:C++ examples gina$ ./rectangleError
area: 1718552992
areab: 30
```

Constructor

- A special member function of a class which is automatically called whenever a new object of a class is created.
- It allows the class to initialize member variables or allocate storage.
- They are only executed once, when a new object is created.
- Declaration: like a regular member function; the name matches the class name; no return type (they initialize an object)

Constructor - An example

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    Rectangle(int, int);
    int area(){return width*height; }
};
Rectangle::Rectangle(int a, int b){
      width = a;
      height = b;
}
int main(){
  Rectangle rect(3,4);
  Rectangle rect_b(5,6);
  cout << " rect area: " << rect.area() << endl;
  cout << " rect_b area: " << rect_b.area() << endl;
  return 0;
}
```

Constructor prototype declaration

Constructor definition

Overloading constructors

Overloading constructors

- A constructor can be overloaded with different versions taking different parameters.
- The compiler will automatically call the one whose parameters match the arguments.
- The *default constructor*: A special kind constructor that takes no parameters. It is called when an object is declared but is not initialized with any arguments.

```
Rectangle rectb; // ok, default constructor called
Rectangle rectc(); // Oops!
```

Constructors: An example

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    Rectangle();
    Rectangle(int, int);
    int area(){return width*height; }
};
Rectangle::Rectangle(){
       width = 5;
       height = 5;
}
Rectangle::Rectangle(int a, int b){
       width = a;
       height = b;
}
```

```
int main(){
  Rectangle rect(3,4);
  Rectangle rect_b;
  cout << "rect area:" << rect.area()
       << endl;
  cout << " ect_b area: " << rect_b.area()
       << endl;
  return 0;
}
```

Calling constructors

Calling constructors

- *functional form*: Enclose the arguments of the constructor in parentheses.

```
class_name object_name ( value1, value2, value3, … )
```

- *Single parameter:*

```
class_name object_name = initialization_value;
```

- *Uniform initialization:* Same as the functional form but using braces instead of parentheses. (Optional: an equal sign before the braces.)

```
class_name object_name { value1, value2, value3, … }
class_name object_name = { value1, value2, value3, … }
```

An example

```
#include <iostream>
using namespace std;
class Circle{
    double radius;
  public:
    Circle(double r){radius = r;};
    double circum(){return 2*radius*3.14159265; }
};

int main(){
  Circle foo(10.0); //functional form
  Circle bar = 20.00; // assignment init.
  Circle baz {30.00}; // uniform init.
  Circle qux = {40.00}; //uniform init.

  return 0;
}
```

Constructors: Initialization

- It is mainly a matter of programming style!
- Uniform vs. functional: Braces cannot be confused with function delcarations.

```
Rectangle rectb; // default constructor called
Rectangle rectc(); // function declaration
Rectangle rectd{}; // default constructor called
```

Member initialization in constructors

Member initialization

- When a constructor is used to initialize other members, these members can be initialized directly.
- Initialization is done by inserting, before the contructor's body, a colon (:) and a list of initializations for class members.

```
class Rectangle{
int width, height;
  public:
    Rectangle(int, int);
    int area(){return width*height; }
};
```

```
1. Rectangle::Rectangle(int a, int b){ width = a; height = b; }
2. Rectangle::Rectangle(int a, int b) : width(a) { height = b; }
3. Rectangle::Rectangle(int a, int b) : width(a), height(b) { }
```

Member initialization

```
class Rectangle{
int width, height;
  public:
    Rectangle(int, int);
    int area(){return width*height; }
};
```

```
1. Rectangle::Rectangle(int a, int b){ width = a; height = b; }
```

Classic constructor definition

```
2. Rectangle::Rectangle(int a, int b) : width(a) { height = b; }
```

```
3. Rectangle::Rectangle(int a, int b) : width(a), height(b) { }
```

Constructor definition with member initialization

Destructor

- A member function of a class that deletes an object
- It helps deallocate the memory of an object
- It does not take any arguments and does not return anything
- There cannot be more than one destructor in a class
- Syntax: ~className
- The compiler creates a default desctructor
  - Problem: Dynamically allocated memory or pointer in a class.
    - Solution: Write a destructor to release memory and avoid memory leak (using delete object).

Pointers to classes

Pointers to classes

- Objects can be pointed to by pointers.
- The members of an object can be accessed directly from a pointer by using the arrow operator(->).
- Syntax:

```
class_name * pointer_name;
```

Operators

Expression	
`*x`	Pointed to by x
`&x`	Address of x
`x.y`	Member y of object x
`x->y`	Member y of object pointed to by x
`(*x).y`	Member y of object pointed to by x
`x[0]`	First object pointed to by x
`x[0]`	Second object pointed to by x
`x[n]`	(n+1)th object pointed to by x

Pointers to classes: An example

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    Rectangle(int x, int y): width(x), height(y){};
    int area(void){return width*height; }
};
int main(){
  Rectangle rect(3,4);
  Rectangle * foo, * bar, * baz;
  foo = &rect;
  bar = new Rectangle (5,6);
  baz = new Rectangle[2]{{2,5},{3,6}};
  cout << " rect's area: " << rect.area() << endl;
  cout << " *foo's area: " << foo->area() << endl;
  cout << " *bar's area: " << bar->area() << endl;
  cout << " baz[0] area: " << baz[0].area() << endl;
  cout << " baz[1] area: " << baz[1].area() << endl;
  delete bar;
  delete[] baz;
  return 0;
}
```

Classes- Alternative definitions

- Classes can be defined also with keywords *struct* and *union*.
- Keyword *struct:* Plain data structures; public access by default.
- Keyword *union:* Store only one data member at a time; public access by default.

Exercise 1

- Write a class (call it Student) that contains the following members: 1) First name;2) Last name; 3) Student ID; 4) Grade (private access).
- The class Student should also contain the following two member functions (public access): 1) storeData():Stores the details of a student (fname, lname, etc..); 2) printData():Prints the details of a student.

Your program should store the details of 3 students (given as input by the user) and then print the details of all students.

Exercise 1 (cont.)

```
Student # 1
************************
Enter first name: Georgia
Enter last name: Koutsandria
Enter the id of the student: 12345
Enter the grade of the student: 30

Student # 2
************************
Enter first name: Gabriele
Enter last name: Saturni
Enter the id of the student: 23456
Enter the grade of the student: 29

Student # 3
************************
Enter first name: Christian
Enter last name: Cardia
Enter the id of the student: 34567
Enter the grade of the student: 28

*************All Students*************
**************************************
F.Name      L.Name       ID       Grade
Georgia     Koutsandria     12345     30
Gabriele     Saturni     23456     29
Christian     Cardia     34567     28
```

Exercise 1 - Solution

```
#include <iostream>
using namespace std;
class Student{
        string fname, lname;
        int student_id, grade;
public:
    void storeData();
    void printData();
};
void Student::storeData(){
        cout << "Enter first name: ", cin >> fname;
        cout << "Enter last name: ", cin >> lname;
        cout << "Enter the id: ", cin >> student_id;
        cout << "Enter the grade: ", cin >> grade;
        cout << endl;
}
void Student::printData(){
        cout << fname << "    " << lname << "    "
             << student_id << "    " << grade << endl;
}
```

Exercise 1 – Solution(cont.)

```
int main(){
	Student students[3];
	for (auto i=0;i<3;i++){
		cout << "Student # " << i+1 << endl;
		cout << "**************************" << endl;
		students[i].storeData();
	}
	cout << "**************All Students**************" << endl;
	cout << "****************************************" << endl;
	cout << "F.Name " << "    " << "L.Name " << "    "
	     << " ID " << "    " << " Grade " << endl;
	for (auto i=0;i<3;i++)
		students[i].printData();

	return 0;
}
```

Exercise 2

Redo exercise 1 using class and pointers.

Exercise 2 - Solution

```
#include <iostream>
using namespace std;
class Student{
        string fname, lname;
       int student_id, grade;
public:
    void storeData();
    void printData();
};
void Student::storeData(){
       cout << "Enter first name: ", cin >> fname;
       cout << "Enter last name: ", cin >> lname;
       cout << "Enter the id: ", cin >> student_id;
       cout << "Enter the grade: ", cin >> grade;
       cout << endl;
}
void Student::printData(){
       cout << fname << "    " << lname << "    "
            << student_id << "    " << grade << endl;
}
```

Exercise 2 – Solution(cont.)

```
int main(){
        Student students[3];
        Student *studentsp;
        studentsp = &students[0];
        for (auto i=0;i<3;i++){
                cout << "Student # " << i+1 << endl;
                cout << "*************************" << endl;
                (studentsp+i)->storeData();
        }
        cout << "**************All Students**************" << endl;
        cout << "****************************************" << endl;
        cout << "F.Name " << "    " << "L.Name " << "    "
             << " ID " << "    " << " Grade " << endl;
        for (auto i=0;i<3;i++)
                (studentsp+i)->printData();

        return 0;
}
```

Additional Resources

- http://www.cplusplus.com/doc/tutorial/
- https://en.cppreference.com/w/
- Programming: Principles and Practice Using C++, Bjarne Stroustrup (Updated for C++11/C++14)
- C++ Primer, Stanley Lippman, Josée Lajoie, and Barbara E. Moo (Updated for C++11)